Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

Proof Automation for Typed Finite Sets

submitted by
Alexander Anisimov

submitted on
August 27, 2015

Supervisor
Prof. Dr. Gert Smolka

Advisor
Christian Doczkal, M.Sc.

Reviewers
Prof. Dr. Gert Smolka
Prof. Bernd Finkbeiner, Ph.D.

Declarations

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

Datum/Date Unterschrift/Signature

Abstract

In this thesis we aim to provide proof automation for typed finite sets. To this intent, we consider three tableau calculi describing increasingly large fragments of set theory. The first one allows for reasoning about the basic set structures $(\cup, \setminus, \{\cdot\}, \emptyset)$ and relations. We prove total correctness for it. Our second calculus extends the first one by a powerset operator. It is still terminating, but we are not able to prove completeness anymore. The third calculus is even stronger and can express not only powersets but also separations. This system is no longer terminating. The decidability of the corresponding fragment of ZF set theory with untyped and possibly infinite sets is an open problem. So, it is likely that the decidability of our fragment is open, too.

For the full system we provide an implementation in Coq/Ssreflect. We consider the alternatives of a direct implementation in Ltac and a proof by computational reflection. The latter is appealing as it is often very efficient in practice. Studying it, we implement a naive decider for boolean tautologies and compare it with the standard automation tactics. But, as reflection requires at least termination, we stick to a direct implementation in Ltac for our set automation.

Acknowledgements

I would like to thank my advisor Christian Doczkal who invested an incredible amount of time and effort guiding me through the process of writing my thesis. I am grateful for his support, his helpful advice and the fruitful discussions that made a remarkable contribution to the competition of this work.

Moreover, I would like to thank my supervisor Prof. Gert Smolka for the two intriguing lectures 'Introduction to Computational Logic' and 'Semantics' that awakened my interest in the topic of computational logic. I am thankful for the possibility to write this thesis and for his support in the course this project.

I would like express my gratitude to Prof. Gert Smolka and Prof. Bernd Finkbeiner for reviewing this thesis.

Lastly, I would like to thank my family for their great support and unconditional understanding during the last year.

Contents

1 Introduction **7**
- 1.1 About this thesis . . . 7
- 1.2 Related Work . . . 8

2 Automated Proof Search in Coq **10**
- 2.1 The Coq Basics . . . 10
  - 2.1.1 Gallina . . . 10
  - 2.1.2 Ltac . . . 11
- 2.2 Ltac vs. Reflection . . . 11
- 2.3 The Structure of a Proof by Reflection . . . 12

3 `shandec`: Reflective Proof Automation for Boolean Logic **14**
- 3.1 Boolean Logic in Ssreflect . . . 14
- 3.2 Implementation of `shandec` . . . 14
  - 3.2.1 Step 1: Preparing the goal . . . 15
  - 3.2.2 Step 2: Reification . . . 15
  - 3.2.3 Step 3: Decision Procedure . . . 16
  - 3.2.4 Step 4: Reflection . . . 17
  - 3.2.5 Final Tactic . . . 17
- 3.3 Comparison of `shandec` with Standard Automation Tactics . . . 18
  - 3.3.1 Proof Terms . . . 18
  - 3.3.2 Runtime . . . 19

4 Analytic Tableaux for Typed Finite Sets **21**
- 4.1 Set Representation and Interpretation . . . 21
- 4.2 Basic System . . . 22
  - 4.2.1 Ruleset . . . 23
  - 4.2.2 Termination . . . 25
  - 4.2.3 Completeness . . . 28
- 4.3 Powerset Extension . . . 33
  - 4.3.1 Differences in Language and Ruleset . . . 33
  - 4.3.2 Termination . . . 34
  - 4.3.3 Possible Incompleteness . . . 35
- 4.4 Separation Extension . . . 35
  - 4.4.1 Differences in Language and Ruleset . . . 35
  - 4.4.2 Divergence . . . 36
- 4.5 Downwards Compatibility of the Systems . . . 36

5 Implementation **38**
5.1 Why Ltac? 38
5.2 Structure 38
5.3 Examples 40

Chapter 1

Introduction

1.1 About this thesis

The goal of the present thesis is to provide proof automation for set-theoretical reasoning. We will base our development on an fset library [3] which implements typed finite sets. It is formalized in Coq/Ssreflect. All concepts of Coq, Ssreflect and fset required to understand the presented ideas will be explained upfront. For the moment it suffices to know that Coq is a higher-order logic proof assistant and bases on constructive type theory. The language used to define objects inside its logic is called Gallina. In this language one can only express terminating functions. In order to reason about its logic from outside, Coq uses an untyped functional language called Ltac. In contract to Gallina, in Ltac one can also write nonterminating programs. Ssreflect is a plug-in for Coq and comes with an amount of useful libraries. In the last few years it has become increasingly prominent. Although it makes extensive use of boolean properties, to the best of our knowledge, there is very little proof automation for it.

Therefore, we decided to investigate this area before starting the set-theoretical part. After all, the fset library is formalized in Ssreflect and hence, requires propositional reasoning, too. Moreover, this investigation gives us the possibility to study a technique called proof by computational reflection based on an example relevant for our purposes. The basic idea, as it is described by Chlipala [1] and Boutin [2], is to take as proof the decision of a certified Gallina procedure. Due to the constant size proof terms it produces, this technique is often highly efficient in practice. It is hence a natural thing to ask, whether it can be applied to the set-theoretical part, too, and what we would need to do so.

Our approach for the set-theoretical part is to design a hierarchy of three tableau systems for finite typed sets. The first system treats the empty set, a denumerable set of variables, singletons, unions and intersections. For this tableau we prove termination and completeness implying that every saturation strategy on it is a decision procedure. Up to this point, a reflective proof approach would be perfectly fine. Our second tableau subsumes the first one and extends it by the powerset operator. The resulting system is strictly stronger. With some minor changes we can carry over the proof of termination but the

proof of completeness breaks. In a setting with untyped and infinite sets, there are still complete and terminating tableau systems for this fragment of set theory. However, the proof of completeness is then highly involved. Still, for our fragment we could use proof by reflection as it only requires termination. But, the fragment we are interested in is even larger. The third tableau extends the second one by the separation operator i.e. we can reason about sets of the form $\{x \in A \mid px\}$ for some set A and some predicate p. We will show that this allows for diverging saturation strategies. Making use of such constructs, the language of the third system is able to express typed universal quantifiers. According to [7] the decidability of an untyped fragment of set theory with infinite sets, capable of expressing universal quantification is an open problem. It is not completely clear whether the typed finite setting is really easier than this. In fact, our setting may even turn out to be harder than the one with the untyped and infinite sets. A prominent example for such a case is the validity of first order logic. While the general setting is semi-decidable its restriction to finite models is not.

Based on these three tableaux we develop a proof search algorithm and implement it in Ltac. Our goal in doing so is to provide an elegant and easy to use solution for set theoretical reasoning. Most of the work currently done considers general sets, i.e. untyped and possibly infinite. But as typing plays an important role in automated provers and proof assistants, it is worthwhile considering this special case. The fset library of Christian Doczkal [3] aims to provide a toolkit for this kind of reasoning and will serve for us as background theory.

1.2 Related Work

The majority of the papers on proof automation in set theory use the language of Multilevel Syllogistic introduced originally by Ferro et al. [4] or some extension of it. The most prominent extension is the so called Multilevel Syllogistic with Singletons (MLSS) which was already suggested in the original paper. The Syllogistic treats untyped sets and admits infinite ones. The terms that can be expressed by MLSS consist of the empty set, a denumerable set of variables and constructors for singletons, unions and set differences. The set of formulas expressible by MLSS is the propositionally complete closure of the membership and equality relation. The underlaying semantics is (in all of the cited papers) Zermelo-Fraenkel set theory. Especially, to the best of our knowledge, proof automation in set theory, so far, has been studied in the untyped setting only. It is therefore non-obvious whether and how the results made in this area can be adapted to our setting.

Cantone and Zarba [5] present a tableau based decision procedure for expressions stated in MLSS. Their focus is on efficient saturation and avoiding the interleaving of model checking steps.

Beckert and Hartmer [6] provide another tableau decision procedure for MLSS. The given system is very similar to the one of those we will present in this thesis. Apart from the fact that we are in a typed, finite setting, the fragment of set theory considered by [6] coincides with the one we treat in our first system.

Cantone and Ruggeri Cannata [9] extend MLSS with an explicit *finite* pred-

icate capturing the resulting fragment of set theory in a set of tableau rules. They prove their calculus decidable, sound and complete.

In 2005 Cantone et al. published a new paper [10] where they provide a decidable, sound and complete tableau calculus for MLSS extended with an iterated membership predicate denoting the transitive closure of the element relation.

The probably closest work to ours is the work of Shults [11]. In his technical report he presents a partial decision procedure for a fragment of set theory larger than the one captured by MLSS and formalized in another language. In contrast to all the other sources his focus lies not on completeness but on expression strength and efficiency. In particular, his system is able to express separations. These two facts constitute a strong overlap of the work done in [11] and the current thesis. The author doesn't fix any semantics emphasizing the flexibility of his system to be able to deal with different semantics, leaving it up to the user to ensure consistency. However there is a restriction in the form of a stratification into pairs of levels which also has certain similarities to the type induced restriction in our system. Another analogy between [11] our system is the handling of set equality. In [11] equality is handled by the system with the aid of extensionality rules and a restricted form of substitution whereas our set representation is inherently extensional and substitution is used whenever a variable can be completely eliminated.

Chapter 2

Automated Proof Search in Coq

In this chapter we will outline the basic features and concepts we will require to write automation tactics in Coq. To this intent we will take a brief look at the languages Gallina and Ltac.

2.1 The Coq Basics

The proof assistant Coq implements the so called calculus of inductive construction and exploits the Curry-Howard-Correspondence. This allows for an association of mathematical propositions with data types. In particular, a proposition is represented as a type and every inhabitant of the type denotes a proof of the proposition.

Let us take now a closer look at the languages that we will use in the following.

2.1.1 Gallina

Gallina is a strictly typed language used to define objects inside the logic in Coq. A Gallina term can be either a constant, a function or a type. Every Gallina term has a type, even if it is one itself. Ill-typed expressions cannot be stated in Gallina. As we shall see in Chapter 5, this property of the typing mechanism will have a significant influence on which fragment of set theory our systems describe.

Function and Dependant Types

Let us now take a closer look at two special kinds of Gallina types that will be of major importance for the automation approaches presented later in this chapter. Let t_1 and t_2 be Gallina types. A so called function type would combine them in the following manner:

$$t_1 \to t_2$$

Its members are functions that on input of an argument of type t_1 return a term of type t_2. As mentioned earlier, Gallina functions are always total.

A dependant type is the generalization of a function type and has the following form:

$$\forall x : t_1.\ t_2$$

Such a type is called dependant because t_2 may depend on the bound variable x. A member of this type would be a function that on input of some $y : t_1$ returns a term of type $t_2{}^x_y$. Note that the dependant type behaves exactly like the function type if there is no free variable x in t_2.

2.1.2 Ltac

We use Gallina to define and compute with terms of the logic underlying Coq. All of these terms must be well-typed. However, we would like to be able to reason about them, as well as about the logic itself, from outside. This is where the language Ltac comes into play. In contrast to Gallina, Ltac is completely untyped. The only distinction it makes is between Ltac and Gallina terms.

One useful feature of Ltac is the more fine grained *match*.

Example. Consider the environment $(a\,b\ :\ bool)$.
While we can match a term like $(a\ \&\&\ b)$ in Gallina only against *true* and *false*, Ltac allows us to match against the term structure itself. For example we could handle conjunctions and disjunctions differently.

Another feature we will often make use of is the possibility to analyze the goal and to extract subterms of it. Especially in combination with the strong pattern matching it is a very powerful tool.

But the probably most important thing about Ltac is the fact that it allows for general (non-structural) recursion. This gives us the possibility to express diverging procedures.

2.2 Ltac vs. Reflection

Using these concepts and mechanisms, Coq allows for several approaches in proof automation. The first one is is the direct implementation of a proof search algorithm in Ltac. The second one is a technique called proof by computational reflection [2]. The high-level idea is to provide a certified Gallina decision procedure and to use the type of its correctness proof. In his Book "Certified Programming with Dependent Types" Adam Chlipala ascribes great importance to this technique. In fact it is one of the main reasons for him to favour Coq over other systems.

The fact that the language Ltac is untyped makes it rather flexible and allows for easy and quick implementation of simple proof search algorithms. Writing such, one has neither to care about termination nor about correctness proofs. However, this convenience comes at the cost of large proof terms and a possibly bad performance. Careful design may solve the performance issue but doesn't help against the large proof terms. In addition, it is hard to implement more involved algorithms in Ltac. Because of the absence of a typing mechanism, debugging Ltac procedures makes a big effort.

Most of these problems don't arise when using reflection. Here, the size of the proof term is always constant in the size of the input. It does not depend on the decision algorithm. These advantages are due to the fact that the algorithm

is implemented in Gallina. However, the technique is not applicable in any case. In order to allow an implementation in Gallina, the decision algorithm has to terminate. Remember that this was not required in Ltac. Furthermore, a proof by reflection consists not only of the decision procedure but also of a reification step and a proof of correctness. Especially the last-mentioned may require, depending on the design of the decision algorithm, a considerable amount of work.

2.3 The Structure of a Proof by Reflection

Let us now take a closer look on how a proof by reflection is constructed. As mentioned earlier in this chapter, our main task in a proof by reflection is to construct a certified decision procedure in Gallina. We can then use its decision together with the proof of correctness to solve our original problem. In general a proof by reflection consists of the following steps:

Syntax Fragment Selection In order to be able to reason in Gallina, we have to provide an inner-logic representation of our problem. However, in this representation we don't necessarily have to be able to express every instance of the problem we are interested in. Instead we can prepend a normalization step ensuring the intermediate problem to fulfill certain form constraints. So, the first thing we have to do is to select the syntactic fragment we actually want to translate. A small fragment results in easier decision procedures and correctness proofs. However, oversimplification may cause considerable performance issues as it bypasses the techniques of proof by reflection with all of their benefits.

Reification With our target fragment in mind, we can now start the translation. The inner-logic representation is provided by an inductive type in Gallina. In the following we will call it the *abstract syntax type*. For disambiguation we will refer to its members as *abstract terms* so as not to confuse them with Gallina terms. Apart from a number of constructors that allow us to represent instances of our intermediate problem uniquely up to conversion, the abstract syntax type needs also a constructor for variables. Gallina terms we want to abstract from or cannot interpret can then be collected in a duplicate-free list and represented as abstract variable terms using their positions.

Decision Procedure Next we design a decision procedure computing on instances of the abstract syntax type in Gallina.

Reflection Lastly we have to prove a correctness lemma that, when applied to an abstract term and a decision of our procedure, yields a proof of the original problem.

Let us make it more precise by having a look at the generic types of the objects described above. Let T be our abstract syntax type, *dec* our decision procedure, *denote* a translation function mapping from abstract to Gallina terms

and *refl* their refection predicate. Consider the following environment:

$$decide : T \to \mathbb{B} \tag{2.1}$$

$$denote : (\mathbb{N} \to Prop) \to T \to Prop \tag{2.2}$$

$$dec\_correct : \forall (s : T)(\phi : \mathbb{N} \to Prop).\ (decide\, s = true) \to denote\, \phi\, s \tag{2.3}$$

The decision procedure has the type given in 2.1. It takes an abstract term and returns *true* if the proposition it represents holds and *false* otherwise. More interesting is the Type 2.2 of the *denote* function. It's task to reconstruct a logical proposition (a member of *Prop*) that is convertible to our normalized problem. To make this possible we need to provide it not only the abstract term but also a function able to recover the uninterpreted syntax from the abstract variables. Such a function has the type $\mathbb{N} \to Prop$ and would simply return the list element at a given position. Recall that we store all of the uninterpreted syntax in a duplicate-free list during the reification step. Lastly, we have the type of the correctness lemma 2.3. It depends on the abstract term and on a variable mapping. Provided the decision procedure evaluates to *true*, Type 2.3 guarantees *dec_correct* to return a proposition convertible to our normalized problem. We have this assertion because the conclusion of the correctness lemma is the fully applied *denote* function.

Proving the correctness lemma is the most important and usually also the most difficult part of a proof by reflection. But, the size of a proof generated via *dec_correct* grows linearly with the size of the abstract term s.

In order to return a proof of $denote\, \phi\, s$ the *dec_correct* needs a proof of $decide\, s = true$ as argument. It can be obtained with a simple Lemma stating that $true = true$. To verify that this is indeed a proof for $decide\, s = true$, Coq will check whether $decide\, s$ reduces to *true*. If the decision procedure happens to return *true* then *dec_correct* has everything it needs to construct the wanted proof term. Otherwise the reduction results in comparing $false = true$ and can not provide *dec_correct* the necessary premise. This way our correctness lemma returns a valid proof for a proposition accepted by the decision procedure and fails on a proposition rejected by it.

Note that such a correctness lemma allows us to use for *decide* not only decider functions but also incomplete proof procedures, as long as they terminate. Termination is required since *decide* is a Gallina function. However it is unproblematic to return $false$ on the abstraction of a valid proposition. Then, the proof would fail in comparing $decide\, s = true$ which would evaluate to $false = true$. Returning *true* on the abstraction of an invalid proposition, in contrast, is not possible. In this case, there would be simply no way to prove correctness.

Chapter 3

`shandec`: Reflective Proof Automation for Boolean Logic

In this chapter we will see an example for the application of a proof by reflection. To this intent, we will elaborate a tactic detecting tautologies in propositional logic. Although the approach of Shannon expansion is very naive, there is quite a number of instances where the reflective procedure outruns the standard automation tactics like `tauto` and `firstorder` in terms of runtime and especially in terms of proof term size.

3.1 Boolean Logic in Ssreflect

The fragment of Coq syntax we want do provide automation for is captured by the following language.

Definition 3.1 (Target Fragment of Coq Syntax)**.**

$$\begin{aligned} form ::= {} & true \mid false \mid x \mid \sim\!\sim form \\ & \mid form \;\&\&\; form \mid form \;||\; form \\ & \mid form \to form \mid form == form \end{aligned}$$

3.2 Implementation of `shandec`

Before starting the reification, we have to determine an appropriate candidate for the abstract syntax type. On the one hand, it has to be strong enough to express the language specified in Definition 3.1. On the other hand it has to be reasonably simple, so that the computations of the decider function and especially the correctness proof don't get out of hand. To this intent, we perform several normalization steps on the goal before applying the techniques of proof by reflection.

3.2.1 Step 1: Preparing the goal

In this normalization step we pursue three main objectives: First, we want to integrate as much information as possible into the formula on which our reflective tactic will operate. Second, we want to decompose complex structures, such that the abstract syntax type only has to cover the more basic cases. And lastly, we want to isolate and neutralize all parts of the original goal that don't provide us any useful information.

We begin this transformation by moving all premisses into the assumptions. Then we replace all Coq equalities on the type bool by boolean equivalences. Afterwards every boolean assumption is reverted and the resulting Coq implication is immediately converted into a boolean one. Repeated rewriting of two simple lemmas eliminates all boolean implications and equalities as described above. Thus, we end up with a goal whose conclusion is a boolean formula without implications and equivalences. The information of all former boolean assumptions and premisses is contained in this formula.

3.2.2 Step 2: Reification

Having normalized the goal as described in the previous section, we can now choose a rather simple abstract syntax type:

Listing 3.1: Abstract Syntax Type

```
1  Inductive term :=
2    | Var: nat → term
3    | TT: term
4    | FF: term
5    | And: term → term → term
6    | Or: term → term → term
7    | Not: term → term.
```

The choice of the abstract syntax type ultimately fixes the target space for the reification process. Now that we have it, we can start reifying the goal by collecting all uninterpretable subterms. As this collection will serve as mapping between the variables and the terms they abstract from, it has to be duplicate free. The induced mapping, will be the correspondence between a term and its position in this collection. This way, syntactically equal terms will be mapped to the same variable. Let us take a look at a simple example illustrating the described scheme.

Example. Let $b : \mathit{bool}$ and $f : \mathit{bool} \to \mathit{bool}$. Consider the following tautology:

$$(b \mid\mid \sim\sim b) \mid\mid f\, b$$

Constructing a collection using the above scheme would result in the list $[b,\ f\, b]$. Neither b nor $f\, b$ consist of structures that can be modeled by any but the first constructor of the abstract syntax type. Thus, both of them are added to the collection. Although there are several occurrences of the variable b in the term, due to the duplicate freeness constraint, it is added to the list exactly once. Moreover, the third occurrence of b is "guarded" by an application of f. As f itself already cannot be interpreted, $f\, b$ is not even seen as an occurrence of b by the algorithm.

Having such a collection, we can now construct an instance of the abstract syntax type that models the formula. Every construct can either be interpreted and expressed by one of the constructors or is contained in the list. Let us illustrate this, again, with an example.

Example. Consider the same setting as in the previous example. Recall the formula $(b \mid\mid \sim\!\sim\ b) \mid\mid f\ b$ and the list $[b,\ f\ b]$. The instance of the abstract syntax type for this formula would be:

```
Or (Or (Var 0) (Not (Var 0))) (Var 1)
```

The indices of the list still can be used to recover the syntax we abstracted from and the numbers below the `Var`-constructors indicate their position in the abstract term. Moreover, as the list is duplicate free, we represent the same fragment with the same variable.

Note that omitting the duplicate freeness constraint would result in a severe loss of expressive power. In fact not even this simple example could be proven to be a tautology anymore.

3.2.3 Step 3: Decision Procedure

The base for our decision procedure is the so called Shannon expansion. It states the following equivalence:

$$tautology(s) \Leftrightarrow \forall x \in vars(s).\ tautology(s^x_{true}) \wedge tautology(s^x_{false})$$

where $tautology(\cdot)$ denotes that a formula is a tautology, $vars(\cdot)$ is the set of all free variables in a formula and s^x_y is the formula s where every occurrence of x is substituted by y.

Iterating this statement over all variables we can reduce the initial formula to a conjunction of closed formulas. For these we can easily decide whether they are valid or not. Apparently the size of the conjunction grows exponential in the number of variables. This fact by itself is not yet surprising as SAT is reducible to our problem, i.e. $(satisfiable(s) \Leftrightarrow \neg tautology(\sim\!\sim\ s))$. Nevertheless we would like to optimize it in a way, such that we have at least in some cases a polynomial runtime. Our first step in this direction is to invoke a simplification function reducing formulas in the canonical way. E.g. The disjunction of a subformula with the constant *true* is equivalent to *true*. In such a case the subformula needn't to be evaluated in order to decide whether the formula is a tautology.

As mentioned before such a simplification function as well as the rest of the decision procedure is written in Gallina. At this point we benefit from working in Coq as it provides rather strong tools for computation and especially we benefit from the reflective proof technique since we simply perform computations on an inductive type and have no need to push around lengthy proof terms.

In order to speed up the procedure even more we might consider the order in which we want to branch on the variables. The benefits we get from invoking a simplification function can be significantly increased, if we choose those variables for branching that would lead to a better result of the simplification. A reasonable heuristic for this, is the number of occurrences. We will branch on the variables that occur the most often first. This way we prune more branches,

increasing the probability to prune a bigger subtree. To this intent we can collect the abstracted syntax fragments during the reification step not only in an ordered and duplicate-free list but also ensure that it is sorted. This can be achieved by storing the numbers of occurrences together with the syntax and reordering the list when it is required.

Having started a this approach in our implementation we had to realize that the computational costs coming with the outlined list operations have a significant part in the overall runtime of the tactic. Although there are problem instances where the tactic terminates in a reasonable time only if we optimize the branching order, they seem rather artificial. In all other cases considering the branching order leads to a notable slow down. In the final version we therefore refrained from optimizing the branching order.

Summing up, our decision procedure performs alternating Shannon expansion and simplification.

3.2.4 Step 4: Reflection

The two main constructs we use for our reflection are the *denote* function and the lemma *taut_denote*.

$$denote : (\mathbb{N} \to \mathbb{B}) \to term \to \mathbb{B}$$
$$taut\_denote : \forall (s : term)\,(phi : nat \to bool).\ taut\, s \to denote\, \phi\, s$$

The function constructs a proposition in our target syntax, i.e. a boolean proposition, out of an abstract term. Such a translation has to invert the steps performed in the reification phase. So, it reconstructs the term structure, plugs back the uninterpreted syntax and yields a proposition convertible to the intermediate problem.

The lemma *taut_denote* proves the correctness of the decision procedure and the mutual correction of the reification and the denote function. Finally, it yields a proof for *denote* ϕ *s*. If ϕ is the variable mapping and s the abstract term representation of a proposition P, *denote* ϕ *s* is convertible to P. Thus, every proof for *denote* ϕ *s* is also a valid proof for P.

3.2.5 Final Tactic

The four steps described above are everything we need to construct reflective proofs for boolean tautologies. We now plug them together to construct the final automation tactic.

Listing 3.2: `shandec`

```
1  Ltac shandec :=
2  preproc;                          (* normalization *)
3  match goal with [ |- ?G = true] =>
4  let xs := allVars (@nil bool) G in
5  let s := translate xs G in
6  exact (taut_denote
7    s                               (* abstract term *)
8    (fun n => nth false xs n)       (* variable maping *)
9    (eq_refl true)                  (* taut s n = true *)
10 )
11 end.
```

We start by moving all the relevant information into the conclusion, as described in Section 3.2.1. This is done by the tactic `preproc` in line 2. The reification is performed in lines 4 and 5. The tactic `allVars` collects all of the uninterpreted boolean terms descending recursively in the interpreted structure of the goal. As it is written in Ltac we can easily match on the boolean connectives in `G`. In the same manner the tactic `translate` constructs the abstract term *s*. It is passed the list of uninterpreted syntax as argument, so it can construct the abstract variables with the correct indices. The rest of the proof is the result of the lemma *taut_denote*. It is applied to the abstract term, a mapping that recovers uninterpreted syntax fragments and a proof of (*true* = *true*) and returns a proof of *denote phi s*. Provided *taut s* returns *true*, the proof can be finished by an `exact` as *denote phi s* is convertible to the current goal.

3.3 Comparison of `shandec` with Standard Automation Tactics

One of the reasons for us to consider automation for boolean logic was the fact that the fset library [3] and Ssreflect make extensive use of boolean properties. However, for this kind of reasoning there isn't much automation in Coq available yet.

A simple but notably suboptimal automation approach is to translate a proposition from `bool` to `Prop` and solve it with one of the common tactics `tauto` or `firstorder`.

This approach as well as the decision procedure for our proof by reflection are quick and dirty implementations. To obtain an efficient reflective automation tactic, one will need a smarter decision procedure. But also the direct Ltac approach can become magnitudes faster if one works directly on `bool` and omits the translations. After all the tactics `tauto` and `firstorder` are designed for intuitionistic reasoning. Although, there exists a possible work-around to use them in described way, they won't be able to exploit the most basic properties in classical reasoning.

The mentioned work-around uses the fact that a proposition is classically provable if and only if its double negation is intuitionistically provable. So, for the translation from `bool` to `Prop` one can write the boolean proposition as its double negation (to which it is equivalent) and then translate it. Thus, if the initial proposition was provable classically, its translation is now provable intuitionistically and we can use `tauto` and `firstorder` to reason about it.

To visualize the conceptional advantages of proof by reflection, let us now compare the proof terms we obtain from either of these automation approaches.

3.3.1 Proof Terms

Let us for now forget about the overhead resulting from the translation and only compare the proof terms that result from `shandec` and `firstorder`.

Listing 3.3: `firstorder` Proof Term

```
1 Example E00 a b: ∼∼ ((a ∨ ∼ a) ∧ (b ∨ ∼ b)).
2 firstorder. Show Proof. Qed.
3
```

```
(fun (a b : Prop) (H : ~ ((a ∨ ~ a) ∧ (b ∨ ~ b))) ⇒
 (fun H0 : a ∨ ~ a → b ∨ ~ b → False ⇒
  (fun (H1 : a → b ∨ ~ b → False)
   (H2 : ~ a → b ∨ ~ b → False) ⇒
   (fun H3 : b ∨ ~ b → False ⇒
    (fun H4 : b ∨ ~ b → False ⇒
     (fun (H5 : b → False) (H6 : ~ b → False) ⇒
      (fun H7 : False ⇒ H7) (H6 H5))
      (fun H5 : b ⇒ H4 (or_introl H5))
      (fun H5 : ~ b ⇒ H4 (or_intror H5))) H3)
    ((fun H3 : a → False ⇒ H2 H3)
       ((fun (_ : False → b ∨ ~ b → False) (H4 : a) ⇒
         (fun H5 : b ∨ ~ b → False ⇒
          (fun (H6 : b → False) (H7 : ~ b → False) ⇒
           (fun H8 : False ⇒ H8) (H7 H6))
            (fun H6 : b ⇒ H5 (or_introl H6))
            (fun H6 : ~ b ⇒ H5 (or_intror H6))) (H1 H4))
         (fun H3 : False ⇒ H2 (fun _ : a ⇒ H3))))))
    (fun H1 : a ⇒ H0 (or_introl H1))
     (fun H1 : ~ a ⇒ H0 (or_intror H1)))
   (fun (H0 : a ∨ ~ a) (H1 : b ∨ ~ b) ⇒ H (conj H0 H1)))
```

Listing 3.4: `shandec` Proof Term

```
Example E01 a b: ((a || ~~ a) && (b || ~~ b)).
shandec. Show Proof. Qed.

(fun a b : bool ⇒
  shandec_denote
    (fun n : nat ⇒ nth false [:: b; a] n)
    (And (Or (Var 1) (Not (Var 1))) (Or (Var 0) (Not (Var 0))))
    (eqxx (T:=bool_eqType) true))
```

The main difference between these two terms is not their size but the way it depends on the size of the problem. While the proof term of `firstorder` grows exponentially with the size of the proposition it is applied to, the size overhead of `shandec` is constant beyond the size of its input! One can explain the growth of the proof term of `firstorder` as follows. Every step of the proof search is performed in Ltac. But, as the problem cannot be solved in a polynomial number of steps, the proof term has to grow at least exponentially. Looking at the proof term of `shandec` we see that the only subterms depending on the input are (`And(Or(Var1) (Not(Var1))) (Or(Var0) (Not(Var0)))`) and [:: `b`; `a`]. The first one is the reification of the proposition and grows linearly in the size of the input. And so does the latter. All the rest stays the same for every possible input.

3.3.2 Runtime

Let us now take a look at the runtime of `shandec` compared with `firstorder` and `tauto`. The runtime of `tauto` and `firstorder` was measured on the already translated goal.

As expected, `tauto` and `firstorder` have their difficulties with classical reasoning. To prove a conjunction of ten excluded middle statements `tauto`

formula	firstorder	tauto	shandec
$\bigwedge_{i=0}^{9}(a_i \vee \neg a_i)$	2.85s	>60s	4.8s
$\left(a_0 \wedge \bigwedge_{i=0}^{60}(a_i \to a_{i+1})\right) \to a_{61}$	3.76s	5.93s	2.04s
$\left(a_0 \wedge \bigwedge_{i=0}^{40}(a_i \to a_{i+1})\right) \to b$	6.44s	3.85s	0s
$\left(\bigwedge_{i=0}^{40}((a_i \to a_{i+1}) \wedge (\neg a_i \to a_{i+1}))\right) \to a_{41}$	2.46s	8.99s	2.70s

The code of these and other examples can be found in tautest.v.

Table 3.1: Runtime Comparison `firstorder - tauto - shandec`

needs more than one minute while `shandec` takes only 4.8 seconds. Still, it is outrun by `firstorder` which takes less than 3 seconds. Another interesting point is the time the tactics need to fail on a non-tautology. The third example is not a tautology. To detect this, `firstorder` needs 6.44 seconds and `tauto` still 3.85. `shandec` does this instantaneously. Summarizing, we can say that despite its naive implementation, `shandec` still can have a decent runtime and in some cases even outruns `tauto` and `firstorder`.

Chapter 4

Analytic Tableaux for Typed Finite Sets

We come now to the main objective of this thesis: the proof automation for finite sets. But before we can start thinking about which technique to use and how to implement it, we have to fix the systems we want to work in. We will start with a simple system for a typed, stratified version of MLSS. Then we will present two extensions to it obtaining strictly stronger systems. However, these extensions will not come for free but each at the cost of a useful property. Let us start with a formal characterization of our interpretation and representation of a set.

4.1 Set Representation and Interpretation

The theory we develop our automation tactics for is the fset library [3] for typed finite sets. So our interpretation of sets should coincide with the one in fset, and so does the representation. Let us now briefly outline the toplevel ideas of fset.

As mentioned before, sets as they are represented in [3] are typed and finite. Every member of a set has to be of the same type and this type must have a certain structure. This is due to the fact that sets are realized as lists over *choiceTypes*. As explained in the Ssreflect documentation, *choiceType* is an "interface for types with a choice operator." The operator is required for the theory behind fset. It is used to obtain an extensional set representation. For the kind of automation we aim to provide, however, we need a type with decidable equality (*eqType* in Ssreflect). Luckily every *choiceType* is also an *eqType*.

Definition 4.1 (*fset*)**.** Let T be a *choiceType*. Then (*fset* T) is the type of finite sets with elements of type T.

For any *choiceType* T, *fset* T is again a *choiceType*. This property allows us to build stratified hierarchies of sets. In such a hierarchy sets of distinct levels have distinct types. This implies that no set or urelement (i.e. a member of a type without toplevel *fset* constructors) can be at two different levels. The level of a set is determined by its type.

Definition 4.2 (Level)**.** We call the number of *fset* constructors in the type of a set its level. We write $lv(s)$ to denote the level of s.

Example. Let T be a type without *fset* constructors. Consider the environment $x : T,\ A : \{fset\ T\}$. Then $lv(x) = 0$ and $lv(A) = 1$.

4.2 Basic System

Our first system is a typed stratified version of MLSS. We can state exactly the same propositions but interpret sets as *fsets* in the way explained above. The following grammar describes, what we can state in our language:

Definition 4.3 (Basic Language)**.**

$$\begin{array}{lll} set & ::= & \dot{\emptyset} \mid x \mid \langle set \rangle \mid set \dot{\cup} set \mid set \dot{-} set \\ rel & ::= & set \dot{\in} set \mid set \dot{\subseteq} set \mid set \doteq set \\ form & ::= & \dot{\bot} \mid rel \mid \dot{\neg} form \mid form \dot{\wedge} form \mid form \dot{\vee} form \mid form \dot{\rightarrow} form \end{array}$$

We call terms of the form *set* set expressions, terms of the form *rel* or their negations relation statements and terms of the form *form* formulas. A ***branch*** is a finite set of well typed formulas.

Assumption. Every relation we will state in this theses is assumed to be well-typed.

Definition 4.4. A branch is called *closed* if it contains $\dot{\bot}$ and *open* otherwise.

Remark. We can represent strict subsets intersections and explicitly given finite sets as follows:

$$\begin{aligned} A \dot{\subset} B &:\Leftrightarrow A \dot{\subseteq} B \dot{\wedge} B \dot{\not\subseteq} A \\ A \dot{\cap} B &:= A \dot{\cup} B \dot{-} ((A \dot{-} B) \dot{\cup} (B \dot{-} A)) \\ \langle x_0, \ldots, x_n \rangle &:= \langle x_0 \rangle \dot{\cup} \ldots \dot{\cup} \langle x_n \rangle \end{aligned}$$

The oversized representation of intersections is definitely nothing one would want to use in an implementation. However, at this point it doesn't cause any trouble since the basic system is only for the theoretical treatment of the topic. In our implementation we will stick to a stronger system which is capable to express both intersections and set differences without any problem.

A branch may contain several formulas, a formula several relation statements, a relation statement several set expressions and a set expression several set literals (i.e. $\dot{\emptyset}$, set variables and urelements). The two last mentioned layers may represent sets of a multitude of distinct levels. Thus, when analyzing a branch, we have to work with a number of objects of distinct types. Seeking automation, we will not get round arranging them in some kind of structure. Fortunately, their types are not completely independent. All of them have the form *fset** T, where a number of *fset* constructors, possibly 0, is applied to some base type T. We can group such objects with respect to their levels.

Definition 4.5. Let Γ be a branch. We define $S_l(\Gamma)$ to be the set of all level-l set expressions occurring somewhere in the term structure of one of the formulas of Γ.

Example. Let T be a type with no toplevel *fset* constructors in it and let $\Gamma := \{x \dot{\in} A \dot{\cup} B\}$ for some $(x : T)$ and some $(A\, B : \{fsetT\})$. Then

$S_0(\Gamma) = \{x\}$
$S_1(\Gamma) = \{A, B, A \dot{\cup} B\}$
$S_2(\Gamma) = S_3(\Gamma) = \ldots = \emptyset$

Since a branch is always a finite set of finite formula terms, the set of its subterms at every level is again finite. For the same reason every branch has only finitely many populated levels.

Fact 4.6. *Let Γ be a branch. Then, $S_l(\Gamma)$ is finite for every $l \in \mathbb{N}$. (Follows from the finiteness of Γ)*

Fact 4.7. *Let Γ be a branch. Then, there exists some $L \in \mathbb{N}$ such that*

$$\forall l \geq L.\ S_l(\Gamma) = \emptyset$$

Definition 4.8. We call the smallest L fulfilling fact above L_Γ.

4.2.1 Ruleset

We will now provide a set of tableau rules which can be used to derive contradictions stated in our language. To prove a proposition, we simply write it as *formula* and find a contradiction to its negation. This suffices for a proof since we all of the relations we can express are decidable.

We search for contradictions by applying the rules illustrated in Figures 4.1, 4.2, 4.3 and 4.4 and thereby extending stepwise our branch. A rule like (S1) with one conclusion is applicable if its premisses are on the branch, but not its conclusion. A rule like (S4) with several conclusions is applicable if at least one of the conclusions isn't on the branch yet. Disjunctive rules like (P2) can be applied if the branch doesn't contain any of their conclusions. Note, that the premisses of the cut rules in Figure 4.3 have a different structure. A cut rule is applicable if both propositions stated in the premisses hold and none of the conclusions is on the branch yet. When applying a non-disjunctive rule, we add its conclusions to the branch. Applying a disjunctive rule, we duplicate the branch and add the conclusions of every side of the disjunction to one of the new branches. No formula is ever removed from a branch.

Definition 4.9. A branch in which no rule is applicable is called *saturated*.

We will present four groups of saturation rules. The propositional rules given in Figure 4.1 apply to complex formulas and deal with their logical connectives. The basic saturation rules of Figure 4.2 are used to infer new relation statements avoiding needless branching. Their main task is to exploit semantic properties of the modeled relations. E.g. the first rule (S1) uses the fact that every member of the subset of some set B is also a member of B itself. The cut rules presented in Figure 4.3 establish new relation statements for existing set expressions. They are capable of relating previously unrelated sets. Thereby, they can provide intermediate steps to detect contradictions that otherwise could not be found. Note that a cut rule can not be applied to a pair of sets that already is connected via the corresponding relation or its negation. Figure 4.4 depicts the branch closing rules. They are used to detect contradictions in a branch. Similar to the basic saturation rules they make use of semantic properties.

Notation. For convenience we will use from now on the following abbreviations:

$$x \dot{\notin} A :\Leftrightarrow \dot{\neg} x \dot{\in} A$$
$$x \dot{\not\subseteq} B :\Leftrightarrow \dot{\neg} A \dot{\subseteq} B$$
$$x \dot{\neq} y :\Leftrightarrow \dot{\neg} x \doteq y$$

(P1) $\dfrac{s \dot{\wedge} t}{s \quad t}$ (P2) $\dfrac{s \dot{\vee} t}{s \mid t}$ (P3) $\dfrac{s \dot{\rightarrow} t}{\dot{\neg} s \mid t}$

(P4) $\dfrac{\dot{\neg}(s \dot{\wedge} t)}{\dot{\neg} s \mid \dot{\neg} t}$ (P5) $\dfrac{\dot{\neg}(s \dot{\vee} t)}{\dot{\neg} s \quad \dot{\neg} t}$ (P6) $\dfrac{\dot{\neg}(s \dot{\rightarrow} t)}{s \quad \dot{\neg} t}$ (P7) $\dfrac{\dot{\neg}\dot{\neg} s}{s}$

Figure 4.1: Propositional Rules

(S1) $\dfrac{x \dot{\in} A \quad A \dot{\subseteq} B}{x \dot{\in} B}$ (S2) $\dfrac{x \dot{\notin} A \quad B \dot{\subseteq} A}{x \dot{\notin} B}$ (S3) $\dfrac{A \dot{\not\subseteq} B}{x_{AB} \dot{\in} A \quad x_{AB} \dot{\notin} B}$

(S4) $\dfrac{A \doteq B}{A \dot{\subseteq} B \quad B \dot{\subseteq} A}$ (S5) $\dfrac{A \dot{\neq} B}{\begin{array}{l|l} x_{AB} \dot{\in} A & x_{BA} \dot{\in} B \\ x_{AB} \dot{\notin} B & x_{BA} \dot{\notin} A \end{array}}$

(S6) $\dfrac{x \doteq y \quad y \dot{\in} A}{x \dot{\in} A}$ (S7) $\dfrac{x \doteq y \quad y \doteq z}{x \doteq z}$ (S8) $\dfrac{x \doteq y}{y \doteq x}$

(S9) $\dfrac{x \dot{\in} \langle y \rangle}{x \doteq y}$ (S10) $\dfrac{x \dot{\notin} \langle y \rangle}{x \dot{\neq} y}$ (S11) $\dfrac{\langle x \rangle \dot{\subseteq} A}{x \dot{\in} A}$

(S12) $\dfrac{x \dot{\in} A \dot{\cup} B}{x \dot{\in} A \mid x \dot{\in} B}$ (S13) $\dfrac{x \dot{\notin} A \dot{\cup} B}{x \dot{\notin} A \quad x \dot{\notin} B}$

(S14) $\dfrac{x \dot{\in} A \dot{-} B}{x \dot{\in} A \quad x \dot{\notin} B}$ (S15) $\dfrac{x \dot{\notin} A \dot{-} B}{x \dot{\notin} A \mid x \dot{\in} B}$

Figure 4.2: Basic Saturation Rules

(C1) $\dfrac{X \in S_l(\Gamma) \quad Y \in S_l(\Gamma)}{X \doteq Y \mid X \dot{\neq} Y}$ (C2) $\dfrac{x \in S_l(\Gamma) \quad A \in S_{l+1}(\Gamma)}{x \dot{\in} A \mid x \dot{\notin} A}$

(C3) $\dfrac{A \in S_l(\Gamma) \quad B \in S_l(\Gamma)}{A \dot{\subseteq} B \mid A \dot{\not\subseteq} B}$

Figure 4.3: Cut Rules

$$\text{(D1)}\ \frac{b \qquad \neg b}{\bot} \qquad \text{(D2)}\ \frac{x \neq x}{\bot} \qquad \text{(D3)}\ \frac{x \in \emptyset}{\bot} \qquad \text{(D4)}\ \frac{x \notin \{x\}}{\bot}$$

Figure 4.4: Branch Closing Rules

The soundness of the given rules is obvious. In addition, for the implementation of our proof search procedure every rule is stated and proven as a lemma in Coq.

The cut rules in Figure 4.3 may cause severe performance issues. If used incorrectly, they can lead to an exponential blow-up without yielding any new information. It is therefore preferable to use them as late as possible. Avoiding them completely, however, is not an option as there exist problems that cannot be solved without them.

Example. Consider the branch

$$A \dot{-} B \dot{\subseteq} \dot{\emptyset},\ x \dot{\in} A,\ x \dot{\notin} B.$$

There is a contradiction because x has to be a member of $A \dot{-} B$ but would then be in $\dot{\emptyset}$. However, none of the basic saturation or branch closing rules is applicable. Somehow, we have to relate the x with $A \dot{-} B$ and the only way to achieve this is via cut rules. Using (C2) we obtain the following tableau:

<table>
<tr><td colspan="3">$A \dot{-} B \dot{\subseteq} \dot{\emptyset}$
$x \dot{\in} A$
$x \dot{\notin} B$</td></tr>
<tr><td>$x \dot{\in} A \dot{-} B$</td><td colspan="2">$x \dot{\notin} A \dot{-} B$</td></tr>
<tr><td>$x \dot{\in} \dot{\emptyset}$
$\dot{\bot}$</td><td>$x \dot{\notin} A$
$\dot{\bot}$</td><td>$x \dot{\in} B$
$\dot{\bot}$</td></tr>
</table>

Every successor branch is closed, so we proved that there is a contradiction in the initial one.

4.2.2 Termination

We will now show that in the basic system every saturation strategy terminates. I.e. no matter in which order we apply the rules, after finitely many steps we reach a point where no rule is applicable anymore.

Wlog, we can assume that we start with a branch without any logical operators except for $\dot{\neg}$. This can be achieved by starting with an arbitrary branch and applying the propositional rules of Figure 4.1 until we have eliminated every logical operator. It is easy to see that every rule eliminates exactly one operator and that for every operator or its negation we have exactly one rule. As soon as all operators are eliminated, we can restrict ourselves to the basic saturation rules in Figure 4.2, the cut rules in Figure 4.3 and the branch closing rules in Figure 4.2. None of these rules would introduce a new logical operator or has one in its premisses. At the same time, none of the propositional rules can be applied anymore since for every instance of a logical connective the conclusion of the corresponding rule is already on the branch. In other words, the saturation can be divided into two independent parts. First is the propositional

saturation which only applies to formulas containing logical operators. Second is the regular saturation that ignores all formulas containing logical operators. The termination of the propositional saturation is rather obvious so we will now concentrate us on the regular saturation.

Assumption. For the rest of this section, we fix Γ to be a branch without binary logical connectives.

For such a branch, we will construct a closure of set expressions. Then, we will prove that it is finite and can conclude that eventually every saturation strategy must terminate.

Definition 4.10 (set expression closure)**.**

$$S_l^+(\Gamma) := \begin{cases} \emptyset & \text{if } l \geq L_\Gamma \\ S_l(\Gamma) \cup f_l(\Gamma) & \text{otherwise} \end{cases} \tag{4.1}$$

$$f_l(\Gamma) := \left\{ x_{uv} \text{ fresh variable at level } l \mid (u,v) \in \left(S_{l+1}^+(\Gamma)\right)^2 \right\} \tag{4.2}$$

$$\mathcal{S}(\Gamma) := \bigcup_{l=0}^{L_\Gamma} S_l^+(\Gamma) \tag{4.3}$$

If we look at the rules that introduce new variables ((S3) and (S5)), we see that for every ordered pair of set expressions (A, B) we introduce at most one new variable $x_{A,B}$. Besides, this new variable is exactly one level below the set expressions that where used to generate it. Therefore, we can show that at every power level we introduce finitely many fresh variables if and only if the next higher level has finitely many set expressions.

Lemma 4.11. $\forall l \in \mathbb{N}.\ S_{l+1}^+(\Gamma)$ *finite* $\Rightarrow f_l$ *finite*

Proof. Let $S_{l+1}^+(\Gamma)$ be finite. Then, $\left(S_{l+1}^+(\Gamma)\right)^2$ is finite, too. By Definition 4.10 we have $|f_l(\Gamma)| = |\left(S_{l+1}^+(\Gamma)\right)^2|$. Thus, $f_l(\Gamma)$ is finite. □

Let us summarize the facts we already know about the level hierarchy:

- By definition $S_{L_\Gamma}^+(\Gamma)$ is empty.
- At every level l, $f_l(\Gamma)$ is finite if $S_{l+1}^+(\Gamma)$ is.
- Since $S_l(\Gamma)$ is finite in any case, $S_l^+(\Gamma) = S_l(\Gamma) \cup f_l(\Gamma)$ is finite at every level $l < L_\Gamma$ where $f_l(\Gamma)$ is finite.

Combining all of these facts, it shouldn't be too difficult to prove that $S_l^+(\Gamma)$ is finite at every level. To this intent we will use the finiteness of the $S_l(\Gamma)$ and propagate inductively the finiteness of the $f_l(\Gamma)$ from level L_Γ to level 0.

Lemma 4.12. $\forall l \in \mathbb{N}.\ S_l^+(\Gamma)$ *is finite.*

Proof. By induction on $n \in \mathbb{N}$ in $l = L_\Gamma - n$

I.B. Let $l \geq L_\Gamma$. By Definition 4.10, $S_l^+(\Gamma)$ is equal to $\emptyset$ and therefore finite.

I.H. $S_{l+1}^+(\Gamma)$ is finite for an $l \in \mathbb{N}$.

I.S. $(l+1 \to l)$ As stated in Fact 4.6, $S_l(\Gamma)$ is finite for every $l \in \mathbb{N}$. From the induction hypothesis we obtain finiteness of $S^+_{l+1}(\Gamma)$ and can conclude via Lemma 4.11 that $f_l(\Gamma)$ has to be finite, too. As a union of finite sets $S^+_l(\Gamma)$ is also finite. □

From the finiteness of $S^+_l(\Gamma)$ for every $l \in \mathbb{N}$ we can easily conclude the finiteness of $\mathcal{S}(\Gamma)$ since it is the union of finitely many $S^+_l(\Gamma)$. From Definition 4.10 it is clear that $\mathcal{S}(\Gamma)$ contains all set expressions occurring in Γ.

Fact 4.13. *$\mathcal{S}(\Gamma)$ is finite as finite union of finite sets.*

Next we will show that $\mathcal{S}(\cdot)$ is closed under application of regular saturation rules.

Lemma 4.14. *Let Δ be a successor branch of Γ after applying one of the saturation rules. Then, $\mathcal{S}(\Delta) \subseteq \mathcal{S}(\Gamma)$.*

Proof. We distinguish 4 cases depending on the applied rule.

1. Let Δ be the result of an application of (S3). Then, $A \dot{\subseteq} B \in \Gamma$ and $\Delta = \Gamma \cup \{x_{AB} \dot{\in} A, x_{AB} \dot{\notin} B\}$. We have

$$\mathcal{S}(\Delta) = \mathcal{S}(\Gamma) \cup \{x_{AB}, A, B\} = \mathcal{S}(\Gamma) \cup \{x_{AB}\}$$

   So, we have to show that $x_{AB} \in \mathcal{S}(\Gamma)$. It holds:

$$\begin{aligned} A, B \in S_{lv(A)}(\Gamma) &\Rightarrow (A, B) \in (S_{lv(A)}(\Gamma))^2 \\ &\Rightarrow x_{AB} \in \left\{x_{uv} \mid (u, v) \in (S_{lv(A)}(\Gamma))^2\right\} = f_{lv(A)-1}(\Gamma) \\ &\Rightarrow x_{AB} \in \mathcal{S}(\Gamma) \end{aligned}$$

   Note, that in the second implication we use that $lv(A) > 0$. We can do this because the relation $A \dot{\subseteq} B$ on the branch requires A and B to have at least level 1.

2. Let Δ be the result of an application of (S5). Then, $A \dot{=} B \in \Gamma$ and either $\Delta = \Gamma \cup \{x_{AB} \dot{\in} A, x_{AB} \dot{\notin} B\}$ or $\Delta = \Gamma \cup \{x_{BA} \dot{\in} B, x_{BA} \dot{\notin} A\}$. We have then either

$$\mathcal{S}(\Delta) = \mathcal{S}(\Gamma) \cup \{x_{AB}\} \quad \textit{or} \quad \mathcal{S}(\Delta) = \mathcal{S}(\Gamma) \cup \{x_{BA}\}$$

   Analogously to case 1. we can show that both x_{AB} and x_{BA} are elements of $\mathcal{S}(\Gamma)$. Hence, $\mathcal{S}(\Delta) \subseteq \mathcal{S}(\Gamma)$ holds in both cases.

3. Let Δ be the result of an application of a cut rule. Then, $\Delta = \Gamma \cup \{x \dot{\circ} y\}$ for some $x, y \in \mathcal{S}(\Gamma)$ and some relation $\dot{\circ}$. Thus, $\mathcal{S}(\Delta) \subseteq \mathcal{S}(\Gamma)$.

4. Let Δ be the result of an application of any other rule. The rule must then have one of the following forms:

$$\frac{a_0 \dot{\circ} a_1 \ \cdots \ a_{n-1} \dot{\circ} a_n}{b_0 \dot{\circ} b_1 \ \cdots \ b_{m-1} \dot{\circ} b_m} \qquad \frac{a_0 \dot{\circ} a_1 \ \cdots \ a_{n-1} \dot{\circ} a_n}{b_0 \dot{\circ} b_1 \mid \cdots \mid b_{m-1} \dot{\circ} b_m}.$$

   We have $\Delta \subseteq \Gamma \cup \{b_0 \dot{\circ} b_1, \ldots, b_{m-1} \dot{\circ} b_m\}$ and hence,

$$\mathcal{S}(\Delta) \subseteq \mathcal{S}(\Gamma) \cup \{b_0, \ldots, b_m\}.$$

It is to show that $\{b_0, \ldots, b_m\} \subseteq \mathcal{S}(\Gamma)$. By construction of the respective rule we have in any case $\{b_0, \ldots, b_m\} \subseteq \{a_0, \ldots, a_n\}$. For the rule to be applicable it is required that

$$\{a_0 \dot{\circ} a_1 \ \cdots \ a_{n-1} \dot{\circ} a_n\} \in \Gamma.$$

Thus, we have

$$\{b_0, \ldots, b_m\} \subseteq \{a_0, \ldots, a_n\} \subseteq \mathcal{S}(\Gamma).$$

□

Using the finiteness and the closure property of $\mathcal{S}(\cdot)$ we can easily proof the termination of the tableau system.

Theorem 4.15. *Every saturation strategy in the basic system terminates.*

Proof. With $\mathcal{S}(\Gamma)$ we have a finite closure of set expressions that can be generated from the initial branch. It is easy to see that every relation statement generated from Γ is of the form $A \dot{\circ} B$ where $A, B \in \mathcal{S}(\Gamma)$ and $\dot{\circ} \in \{\dot{\in}, \dot{\notin}, \dot{\subseteq}, \dot{\not\subseteq}, \doteq, \dot{\neq}\}$. Hence, there are six kinds of binary relations and $|\mathcal{S}(\Gamma)|$ possible arguments. The upper bound for the number of relation statements that can be generated from Γ is therefore $6 * |\mathcal{S}(\Gamma)|^2$. As explained at the beginning of Section 4.2.1 a relation statement is neither removed from the branch nor added to it twice. The application of any rule adds at least one new relation statement preserving the set expression closure property. Altogether the size of the branch grows strictly monotone and is upper bounded by the number $6 * |\mathcal{S}(\Gamma)|^2$. Hence there must be a point when we can't add any new relation statements, so the system has to terminate. □

4.2.3 Completeness

In this section we will show that the basic system is complete. We will begin with a formal definition of completeness and afterwards show that our calculus fulfills it.

For the definition we first explain what we will consider a variable assignment. Then we will lift it up to define a model and finally determine what it means for a model to satisfy a branch. From this we will then construct the definition of completeness.

Definition 4.16 (Variable Assignment)**.** Let $vars_l(\Gamma)$ denote the set of all variables of Γ at level l. We call J a variable assignment if it has the type

$$J : \forall l.\ vars_l(\Gamma) \to fset^l(D),$$

where D is some domain type and $fset^l$ the l-times application of the *fset* type constructor.

From the variable assignment, we can obtain a model via the following recursive construction.

Definition 4.17 (Model)**.** Let J be a variable assignment. We define the *model* induced by J as follows:

$$\begin{aligned}\bar{J}\dot{\emptyset} &:= \emptyset \\ \bar{J}A &:= JA \text{ if } A \in vars_l(\Gamma) \text{ for some } l \in \mathbb{N} \\ \bar{J}\langle x\rangle &:= \{\bar{J}x\} \\ \bar{J}A\dot{\cup}B &:= \bar{J}B \cup \bar{J}C \\ \bar{J}A\dot{-}B &:= \bar{J}B \backslash \bar{J}C\end{aligned}$$

Defining a model we fix the semantics for our set operations. We will use alternative symbols like $\dot{\cup}$ and $\dot{\in}$ to speak about the literals on the branch and the regular symbols like $\cup$ and $\in$ to refer to their semantic properties.

Definition 4.18 (Satisfiability)**.** Let J be a variable assignment and $\bar{J}$ the model induced by it. We say that an assignment/its model satisfies a relation $A \dot{\circ} B$ if we have:

$$J \models A \dot{\circ} B :\Leftrightarrow \bar{J}A \circ \bar{J}B$$

for $\dot{\circ} \in \{\dot{\in}, \dot{\notin}, \dot{\subseteq}, \dot{\not\subseteq}, \doteq, \dot{\neq}\}$ and $\circ$ the corresponding semantic relation. We define satisfiability of formulas as follows:

$$\begin{aligned} J \models s\dot{\wedge}t \quad &:\Leftrightarrow \quad J \models s \wedge J \models t \\ J \models s\dot{\vee}t \quad &:\Leftrightarrow \quad J \models s \vee J \models t \\ J \models s\dot{\rightarrow}t \quad &:\Leftrightarrow \quad J \models s \rightarrow J \models t \end{aligned}$$

Γ is called *satisfiable*, if there exists some J such that for all literals $\phi \in \Gamma$ we have $J \models \phi$.

Remark. If $J \models \Gamma$ then we have for every subset $\Delta \subseteq \Gamma,\ J \models \Delta$.

Definition 4.19 (Completeness)**.** A tableau system is complete, if every branch that remains open after a full saturation is satisfiable.

So, to show completeness we have to prove that every open saturated branch has a model. In other words, if none of our rules is applicable and we still haven't found a contradiction, there is an assignment under which the conjunction of the formulas in the branch holds. The fact, that our branches are finite at any time allows us to do to do this proof constructively.

We will start by defining a domain for our variable assignment and then give an interpretation function for set expressions. This interpretation function will play the role of an assignment on the variables. From the assignment we will lift a model as described in Definition 4.17. We will prove that the so obtained model satisfies every formula on the branch it was defined for. This will yield that every open and saturated branch has a model which is equivalent to the fact that the considered system is complete.

Assumption. We fix Γ to be an open saturated branch.

Remark. The fact that Γ is open and saturated implies that it doesn't contain any contradictory formulas nor it contains pair of a formula and its negation. In both of these cases, one of the branch closing rules would have to be applicable, so Γ could not be saturated. Otherwise $\dot{\bot}$ would be already on the branch and Γ could not be open.

We define an interpretation of the base type and call it domain. It contains all terms at level 0 but doesn't distinguish those which are equivalent to each other.

Definition 4.20 (Domain)**.** The domain of a branch is the set of equivalence classes

$$D_\Gamma := S_0^+(\Gamma)/\dot{=}.$$

In order for D_Γ to be well-defined, $\dot{=}$ has to be an equivalence relation.

Lemma 4.21. *In an open saturated branch* Γ*,* $\dot{=}$ *is an equivalence relation.*

Proof. Reflexivity:
For any $x \in \mathcal{S}(\Gamma)$, either $x \dot{=} x$ or $x \dot{\neq} x$ has to be on the branch due to (C1). If it was $x \dot{\neq} x$ the branch would also contain $\dot{\bot}$ due to (D2). But then Γ wouldn't be open anymore. Hence, $x \dot{=} x \in \Gamma$.
Symmetry:
Assume Γ contains $x \dot{=} y$. By (S8) it must also contain $y \dot{=} x$.
Transitivity:
Assume Γ contains $x \dot{=} y$ and $y \dot{=} z$. By (S7) it must also contain $x \dot{=} z$. □

Starting with D_Γ, we define a level-wise interpretation for sets.

Definition 4.22 (Interpretation)**.** We define the interpretation function as follows:

$$\mathcal{I} : \forall l \in \mathbb{N}.\ S_l^+(\Gamma) \to \mathit{fset}^l(D_\Gamma)$$

$$\mathcal{I}_l(X) = \begin{cases} [X]_{\dot{=}} & l = 0 \\ \{\mathcal{I}_{l-1}(x) \mid x \dot{\in} X \in \Gamma\} & l > 0 \end{cases}$$

Remark. a) The argument l of $\mathcal{I}_l(X)$ can always be inferred from the type of X. In the following, we will therefore suppress it: $\mathcal{I}(X) := \mathcal{I}_{lv(X)}(X)$.

b) The definition of $\mathcal{I}$ depends on Γ.

Before we proceed, let us show a small but useful technical lemma. It states that a well-typed relation between members of $\mathcal{S}(\Gamma)$ is on the branch if and only if its negation is not.

Lemma 4.23. *For all* $X, Y \in \mathcal{S}(\Gamma)$ *and* $\dot{\circ} \in \{\dot{\in}, \dot{\notin}, \dot{\subseteq}, \dot{\not\subseteq}, \dot{=}, \dot{\neq}\}$*,*

$$X \dot{\circ} Y \in \Gamma \Leftrightarrow \dot{\neg} X \dot{\circ} Y \notin \Gamma.$$

Proof. "⇒"Let $X \dot{\circ} Y \in \Gamma$. Then, $\dot{\neg} X \dot{\circ} Y$ can't be an element of Γ since the branch would have been closed by (D1).
"⇐"Let $\dot{\neg} X \dot{\circ} Y \notin \Gamma$. Due to the cut rules of Figure 4.3, either $X \dot{\circ} Y$ has to be an element of Γ or $\dot{\neg} X \dot{\circ} Y$. Since $\dot{\neg} X \dot{\circ} Y$ is not an element, $X \dot{\circ} Y$ has to be. □

We will now show a certain closure property of $\mathcal{I}$. It is very similar to the definition of a model satisfying a branch but should not be confused with it. In particular, our interpretation function as a whole is neither a model nor a variable assignment. So, at this point we cannot speak about satisfiability yet. However, as soon as we have a model this lemma will become very useful. It states, that the interpretations of two sets are in a certain relation to each other, if and only if the corresponding relation statement between these sets is on the branch.

Lemma 4.24. *Let* $X, Y \in \mathcal{S}(\Gamma)$ *and* $\dot\circ \in \{\dot\in, \dot\notin, \dot\subseteq, \dot\not\subseteq, \doteq, \dot\neq\}$. *Then*

$$\mathcal{I}X \circ \mathcal{I}Y \Leftrightarrow X \dot\circ Y \in \Gamma.$$

Proof. Induction on $l = lv(Y)$.

I.B. Let $l = 0$. Then $\dot\circ \notin \{\dot\in, \dot\notin, \dot\subseteq, \dot\not\subseteq\}$.

- $(\doteq)$ $\mathcal{I}X = \mathcal{I}Y \Leftrightarrow [X]_{\doteq} = [Y]_{\doteq} \Leftrightarrow X \doteq Y \in \Gamma$.
- $(\dot\neq)$ By Lemma 4.23 we have $X \dot\neq Y \in \Gamma \Leftrightarrow X \doteq Y \notin \Gamma$. What is left to show is that $\mathcal{I}X \neq \mathcal{I}Y \Leftrightarrow X \doteq Y \notin \Gamma$. Negating both sides of the equivalence we obtain $\mathcal{I}X = \mathcal{I}Y \Leftrightarrow X \doteq T \in \Gamma$. And this is exactly what we have shown in case $(\doteq)$.

I.H. For a y at some level $l \geq 0$ we have

$$\forall x \in \mathcal{S}(\Gamma), \dot\circ \in \{\dot\in, \dot\notin, \dot\subseteq, \dot\not\subseteq, \doteq, \dot\neq\}.\ \mathcal{I}x \circ \mathcal{I}y \Leftrightarrow x \dot\circ y \in \Gamma$$

I.S. $l \to l+1$, $(lv(Y) = l+1)$:

- $(\dot\in)$ "$\Rightarrow$"Let $\mathcal{I}X \in \mathcal{I}Y$. Unfolding Definition 4.22 we obtain $\mathcal{I}X \in \{\mathcal{I}y \mid y \dot\in Y \in \Gamma\}$. Hence, there is a $y \in \mathcal{S}(\Gamma)$ such that $y \dot\in Y \in \Gamma$ and $\mathcal{I}X = \mathcal{I}y$. Since $lv(X) = lv(y) < lv(Y)$, the lest equality yields $X \doteq y \in \Gamma$. But as both $X \doteq y$ and $y \dot\in Y$ are on the branch, so is $X \dot\in Y$ due to (S6). And this is what we had to show.
  "$\Leftarrow$"Let $X \dot\in Y \in \Gamma$. Then, $\mathcal{I}X \in \{\mathcal{I}x \mid x \dot\in Y \in \Gamma\} = \mathcal{I}Y$
- $(\dot\notin)$ $\mathcal{I}X \notin \mathcal{I}Y \overset{(\dot\in)}{\Longleftrightarrow} X \dot\in Y \notin \Gamma \overset{4.23}{\Longleftrightarrow} X \dot\notin Y \in \Gamma$
- $(\dot\subseteq)$ "$\Rightarrow$"Let $\mathcal{I}X \subseteq \mathcal{I}Y$. Since Γ is saturated, it must contain either $X \dot\subseteq Y$ or $X \dot\not\subseteq Y$ due to (C3). Assume towards contradiction that $X \dot\not\subseteq Y \in \Gamma$. By (S3), $x_{XY} \dot\in X$ and $x_{XY} \dot\notin Y$ must then be on the branch. As shown in cases $(\dot\in)$ and $(\dot\notin)$ we then have $\mathcal{I}(x_{XY}) \in \mathcal{I}X$ and $\mathcal{I}(x_{XY}) \notin \mathcal{I}Y$. This contradicts $\mathcal{I}X \subseteq \mathcal{I}Y$. Hence, $X \dot\subseteq Y$ must be on the branch.
  "$\Leftarrow$"Let $X \dot\subseteq Y \in \Gamma$. By (S1) we know that for every $x \in \mathcal{S}(\Gamma)$ with $x \dot\in X \in \Gamma$ there is also $x \dot\in Y$ on the branch. Hence, $\{x \mid x \dot\in X \in \Gamma\} \subseteq \{x \mid x \dot\in Y \in \Gamma\}$. Therefore, we have $\mathcal{I}X = \{\mathcal{I}x \mid x \dot\in X \in \Gamma\} \subseteq \{\mathcal{I}x \mid x \dot\in Y \in \Gamma\} = IY$.
- $(\dot\not\subseteq)$ $\mathcal{I}X \not\subseteq \mathcal{I}Y \overset{(\dot\subseteq)}{\Longleftrightarrow} X \dot\subseteq Y \notin \Gamma \overset{4.23}{\Longleftrightarrow} X \dot\not\subseteq Y \in \Gamma$
- $(\doteq)$ $\mathcal{I}X = \mathcal{I}Y \Leftrightarrow \mathcal{I}X \subseteq \mathcal{I}Y \wedge \mathcal{I}Y \subseteq \mathcal{I}X \overset{(\dot\subseteq)}{\Longleftrightarrow} X \dot\subseteq Y, Y \dot\subseteq X \in \Gamma$. So, what is left to show is $X \dot\subseteq Y, Y \dot\subseteq X \in \Gamma \Leftrightarrow X \doteq Y \in \Gamma$.
  "$\Rightarrow$"Let $X \dot\subseteq Y, Y \dot\subseteq X \in \Gamma$. Since Γ is saturated, due to (C1) there must be either $X \doteq Y$ or $X \dot\neq Y$ on the branch. Assume towards contradiction that it is $X \dot\neq Y$. Then, by (S5), there must be either $x_{XY} \dot\in X$ and $x_{XY} \dot\notin Y$ or $x_{YX} \dot\notin X$ and $x_{YX} \dot\in Y$ on the branch. The first case would imply $\mathcal{I}X \ni \mathcal{I}x_{XY} \notin \mathcal{I}Y$ and contradict $\mathcal{I}X \subseteq \mathcal{I}Y$. The second case would imply $\mathcal{I}Y \ni \mathcal{I}x_{YX} \notin \mathcal{I}X$ and contradict $\mathcal{I}Y \subseteq \mathcal{I}X$. Hence, $X \dot\neq Y$ cannot be on the branch, so we have $X \doteq Y \in \Gamma$.
  "$\Leftarrow$"Let $X \doteq Y \in \Gamma$. By (S4), we then have $X \dot\subseteq Y$ and $Y \dot\subseteq X$ on the branch, which is exactly what we had to show.

$(\dot{\neq})$ $\mathcal{I}X \neq \mathcal{I}Y \overset{(\doteq)}{\Longleftrightarrow} X \doteq Y \notin \Gamma \overset{4.23}{\Longleftrightarrow} X \dot{\neq} Y \in \Gamma$ □

Restricting our interpretation function $\mathcal{I}$ to variables we obtain an assignment for Γ.

Definition 4.25. Let $vars(\Gamma)$ be the set of variables of Γ. For our completeness proof we will use the following variable assignment:

$$I = \mathcal{I}|_{vars(\Gamma)}$$

where $\mathcal{I}|_{vars(\Gamma)}$ is the usual function restriction of $\mathcal{I}$ to the domain $vars(\Gamma)$.

In the following Lemma we will show that the model $\bar{I}$ induced by I coincides with $\mathcal{I}$ on sets that are in the closure of Γ.

Lemma 4.26. *In the environment* $(T : choiceType)$, $(x : T)$, $(A, B : fset\ T)$ *the following holds:*

a) $\mathcal{I}\dot{\emptyset} = \emptyset$

b) $\langle x \rangle \in \mathcal{S}(\Gamma) \Rightarrow \mathcal{I}\langle x \rangle = \{\mathcal{I}x\}$

c) $A \dot{\cup} B \in \mathcal{S}(\Gamma) \Rightarrow \mathcal{I}(A \dot{\cup} B) = \mathcal{I}A \cup \mathcal{I}B$

d) $A \dot{-} B \in \mathcal{S}(\Gamma) \Rightarrow \mathcal{I}(A \dot{-} B) = \mathcal{I}A \backslash \mathcal{I}B$

Proof. a) $\mathcal{I}\dot{\emptyset} = \{\mathcal{I}x \mid x \dot{\in} \dot{\emptyset} \in \Gamma\} = \emptyset$ since any literal $x \dot{\in} \dot{\emptyset}$ can be used to close the branch.

b) Let $\langle x \rangle \in \Gamma$. We show $\mathcal{I}\langle x \rangle = \{\mathcal{I}x\}$.
For every literal $y \dot{\in} \langle x \rangle \in \Gamma$ we have also $y \doteq x \in \Gamma$ by (S9). There is at least one literal of this form since we can infer $x \dot{\in} \langle x \rangle$ with (C2) as $\{x, \langle x \rangle\} \subseteq \mathcal{S}(\Gamma)$. For every y with $y \doteq x \in \Gamma$, Lemma 4.24 yields that $\mathcal{I}y = \mathcal{I}x$. Hence, $\mathcal{I}\langle x \rangle = \{\mathcal{I}y \mid y \dot{\in} \langle x \rangle \in \Gamma\} = \{\mathcal{I}x\}$ since all of the $\mathcal{I}y$ are equal to $\mathcal{I}x$.

c) Let $A \dot{\cup} B \in \mathcal{S}(\Gamma)$. We show $\mathcal{I}(A \dot{\cup} B) = \mathcal{I}A \cup \mathcal{I}B$.
"$\subseteq$" Let $s \in \mathcal{I}(A \dot{\cup} B) = \{\mathcal{I}x \mid x \dot{\in} A \dot{\cup} B \in \Gamma\}$. Then, there exists some $x \in \mathcal{S}(\Gamma)$ with $x \dot{\in} A \dot{\cup} B \in \Gamma$ such that $s = \mathcal{I}x$. By (S12) we know that either $x \dot{\in} A$ or $x \dot{\in} B$ is on the branch. So, s is either in $\{\mathcal{I}x \mid x \dot{\in} A \in \Gamma\}$ or in $\{\mathcal{I}x \mid x \dot{\in} B \in \Gamma\}$. Hence, $s \in \mathcal{I}A \cup \mathcal{I}B$.
"$\supseteq$" Let $s \in \mathcal{I}A \cup \mathcal{I}B$. Then s is either a member of $\{\mathcal{I}x \mid x \dot{\in} A \in \Gamma\}$ or a member of $\{\mathcal{I}x \mid x \dot{\in} B \in \Gamma\}$. In the first case, there exists some $a \in \mathcal{S}(\Gamma)$ with $\mathcal{I}a = s$ such that $a \dot{\in} A$ is on the branch. In the second case, there exists some $b \in \mathcal{S}(\Gamma)$ with $\mathcal{I}b = s$ such that $b \dot{\in} B$ is on the branch. Since $A \dot{\subseteq} B \in \mathcal{S}(\Gamma)$, we know by (C2) that $a \dot{\in} A \dot{\cup} B \in \Gamma$ and $b \dot{\in} A \dot{\cup} B \in \Gamma$. Otherwise we had $a \dot{\notin} A \dot{\cup} B$ or $b \dot{\notin} A \dot{\cup} B$ on the branch and could infer either $a \dot{\notin} A$ or $b \dot{\notin} B$ with (S13). Thus, $\mathcal{I}a, \mathcal{I}b \in \{\mathcal{I}x \mid x \dot{\in} A \dot{\cup} B \in \Gamma\}$ and s is in both cases in $\mathcal{I}A \cup \mathcal{I}B$.

d) Let $A \dot{-} B \in \mathcal{S}(\Gamma)$. We show $\mathcal{I}(A \dot{-} B) = \mathcal{I}A \backslash \mathcal{I}B$
"$\subseteq$" Let $s \in \mathcal{I}(A \dot{-} B) = \{\mathcal{I}x \mid x \dot{\in} A \dot{-} B \in \Gamma\}$. Then, there exists some $x \in \mathcal{S}(\Gamma)$ with $s = \mathcal{I}x$ such that $x \dot{\in} A \dot{-} B \in \Gamma$. By (S14) we know that both $x \dot{\in} A$ and $x \dot{\notin} B$ are on the branch. Lemma 4.24 yields that $\mathcal{I}x \in \mathcal{I}A$ and $\mathcal{I}x \notin \mathcal{I}B$. Hence, $s = \mathcal{I}x \in \mathcal{I}A \backslash \mathcal{I}B$.
"$\supseteq$" Let $s \in \mathcal{I}A \backslash \mathcal{I}B$. Then, $s \in \mathcal{I}A$ and $s \notin \mathcal{I}B$. So there exists some

$a \in \mathcal{S}(\Gamma)$ with $s = \mathcal{I}a$ such that $a \dot{\in} A \in \Gamma$ but no b with $s = \mathcal{I}b$ such that $b \dot{\in} B \in \Gamma$. In other words, for every $b \in \mathcal{S}(\Gamma)$, if $\mathcal{I}b = s$ then $b \dot{\in} B$ is not on the branch. In particular, we have $a \dot{\in} B \notin \Gamma$. As $A \dot{-} B \in \mathcal{S}(\Gamma)$, we can infer via (C2) that either $a \dot{\in} A \dot{-} B$ or $a \dot{\notin} A \dot{-} B$ is on the branch. If it was $a \dot{\notin} A \dot{-} B$, due to (S15) there would be either $a \dot{\notin} A$ or $a \dot{\in} B$ on the branch. Any of these relations is contradictory. Hence, we have $a \dot{\in} A \dot{-} B \in \Gamma$ and can conclude via Lemma 4.24 that $s = \mathcal{I}a \in \mathcal{I}(A \dot{-} B)$. □

Summing up, we have shown in Lemma 4.26:

$$\forall A \in \mathcal{S}(\Gamma),\ \mathcal{I}A = \bar{I}A.$$

Having this correspondence, we can now exploit the properties we have shown for $\mathcal{I}$ to prove $\bar{I} \models \Gamma$

Theorem 4.27. *The basic system is complete.*

Proof. Let Δ be an arbitrary initial branch. We distinguish two cases:

1. Every fully saturated successor branch we can obtain from Δ using the saturation rules of our basic system is closed. As our ruleset is sound, this fact by itself is already a proof for the unsatisfiability of Δ.

2. There is an open saturated branch Γ with $\Delta \subseteq \Gamma$. In Definition 4.25 we gave a variable assignment for Γ. We showed in Lemma 4.26 that the model induced by the variable assignment coincides with $\mathcal{I}$ from Definition 4.22 on every set in $\mathcal{S}(\Gamma)$. But for $\mathcal{I}$ we already know that

   $$\forall A, B \in \mathcal{S}(\Gamma).\ \mathcal{I}A \circ \mathcal{I}B \Leftrightarrow A \dot{\circ} B \in \Gamma$$

   from Lemma 4.24. Thus, for all sets in $\mathcal{S}(\Gamma)$ we have

   $$\bar{I}A \circ \bar{I}B \Leftrightarrow A \dot{\circ} B \in \Gamma$$

   which is by definition equivalent to $I \models \Gamma$. Since $\Delta \subseteq \Gamma$, we have also $I \models \Delta$. □

4.3 Powerset Extension

In the last section we presented the basic system. Its language can express the common set operations and logical connectives as well as the relations membership, subset and equality. We provided a proof of total correctness ensuring that every saturation strategy is a decision procedure. Now we will strengthen this system by enriching it with the powerset operator $\dot{\mathcal{P}}(\cdot)$. We will investigate how this affects the properties we have proven for the basic system.

4.3.1 Differences in Language and Ruleset

The language changes as follows:

$$set ::= \ldots \mid \dot{\mathcal{P}}(set)$$

$$\text{(Q1)}\ \frac{A\dot{\in}\dot{\mathcal{P}}(B)}{A\dot{\subseteq}B} \qquad \text{(Q2)}\ \frac{A\dot{\notin}\dot{\mathcal{P}}(B)}{x_{AB}\dot{\in}A \qquad x_{AB}\dot{\notin}B}$$

Figure 4.5: Powerset Rules

We also need some new saturation rules to characterize the semantic properties of powersets:

It is worth noting that the introduction of a powerset operator significantly increases the importance of the cut rules. Although we had already in the basic system problems that were not solvable without cut rules, they are now needed much more frequently. Cut rules become a basic tool to establish relations between different levels.

Example. The following branch has an obvious contradiction.

$$A\dot{\nsubseteq}B,\ \dot{\mathcal{P}}(A)\dot{\subseteq}\dot{\mathcal{P}}(B)$$

A is definitely a member of its own powerset. Since $\dot{\mathcal{P}}(A)\dot{\subseteq}\dot{\mathcal{P}}(B)$, A has also to be a member of $\dot{\mathcal{P}}(B)$. But then it must also be a subset of B, which it is not.

To find this contradiction we must use a cut rule. Otherwise we had no chance to infer any relation that would connect the level of A with the level of its powerset. The membership cut rule (C2) applied to A and $\dot{\mathcal{P}}(A)$, would do this job. Using it, we obtain the following tableau:

$A\dot{\nsubseteq}B$ $\dot{\mathcal{P}}(A)\dot{\subseteq}\dot{\mathcal{P}}(B)$	
$A\dot{\in}\dot{\mathcal{P}}(A)$	$A\dot{\notin}\dot{\mathcal{P}}(A)$
$A\dot{\in}\dot{\mathcal{P}}(B)$	$x_{AA}\dot{\in}A$
$A\dot{\subseteq}B$	$x_{AA}\dot{\notin}A$
$\dot{\bot}$	$\dot{\bot}$

The contradiction could also be inferred with (C2) applied to A and $\dot{\mathcal{P}}(B)$.

This example is not a rare margin case but representative of a large class of problems. If we cannot infer a membership relation between two levels, there is no way to exploit the subset relations on the higher one. However, in the case of powersets this may still be necessary. It is not hard to construct a branch where the subset relations between powersets contradict relations on the levels below.

4.3.2 Termination

To verify that the so obtained system still terminates, we have to show that the new rules preserve the closure property. It is easy to see that for (Q1) we have:

$$\{A, B\} \subseteq \mathcal{S}(\{A\dot{\in}\dot{\mathcal{P}}(B)\}).$$

But also the freshly generated variable of (Q2) is still inside the closure:

$$\{x_{AB}, A, B\} \subseteq \mathcal{S}(\{A\dot{\notin}\dot{\mathcal{P}}(B)\})$$

One can show this analogously to the cases 1 and 2 of the proof of Lemma 4.14.

4.3.3 Possible Incompleteness

Consider the following branch:

$$\Gamma := \{\dot{\mathcal{P}}(A)\dot{-}\langle A\rangle \mathrel{\dot{\subseteq}} \dot{\emptyset}, \quad x\dot{\in}A\}$$

This is a contradiction because the only set whose powerset contains only itself is the empty set. But then it could not contain any element. The only way to bring $x\dot{\in}A$ to a contradiction is to show $A\dot{\subseteq}\dot{\emptyset}$. But, as we are in a typed setting, we must therefore have $lv(\dot{\emptyset}) = lv(A)$. The $\dot{\emptyset}$ that in the branch, in contrast, is on the level of $\dot{\mathcal{P}}(A)$ and thus, one level above A. So we have $\dot{\emptyset} \notin S_{lv(A)}(\Gamma)$ and can therefore can not apply (C3). If we replace the second literal with $A\dot{\neq}\dot{\emptyset}$, however, the branch can be closed without any problems.

4.4 Separation Extension

So far we have presented two tableau calculi for typed finite sets. The first one is terminating and complete, but rather weak when it comes to expressive power. It is only able to express the empty set, variables, singletons, unions, and differences. The second calculus is stronger in the sense that it additionally allows for the powerset operator. It is still terminating but may no longer be complete.

Now we would like to strengthen our language even further. For this purpose we introduce the separation constructor $\langle x \in A \mid p\rangle$. Apart from its frequent usage, there is one more reason for us to consider this operator. As we will see in the following section, it can be used to express intersections and differences. In particular, this is how these operators are defined in the fset library. We will also see, that this extension doesn't come for free but at the cost of termination. Although a terminating system would certainly desirable, to the best of our knowledge, the decidability of the considered fragment of set theory is an open question.

4.4.1 Differences in Language and Ruleset

We extend our language by the separation operator:

$$\langle x\dot{\in}set \mid form\rangle$$

This allows us to express intersection, set difference and even universal quantification in terms of separations.

$$\begin{aligned} A\dot{\cap}B &:= \langle x\dot{\in}A \mid x\dot{\in}B\rangle \\ A\dot{-}B &:= \langle x\dot{\in}A \mid x\dot{\in}B\rangle \\ \dot{\forall}x\dot{\in}A.\, p &:\Leftrightarrow A\dot{\subseteq}\langle x\dot{\in}A \mid p\rangle \end{aligned}$$

A set in our new language after adding the separation and removing the difference operators may looks as follows:

$$set ::= \dot{\emptyset} \mid x \mid \langle set\rangle \mid set\dot{\cup}set \mid \dot{\mathcal{P}}(set) \mid \langle x\dot{\in}set \mid form\rangle$$

To be able to deal with this new operator we have to add two new saturation rules (Figure 4.6).

$$\text{(R1)}\ \frac{y\dot{\in}\langle x\dot{\in}A \mid p\rangle}{y\dot{\in}A \qquad p_y^x} \qquad\qquad \text{(R2)}\ \frac{y\dot{\notin}\langle x\dot{\in}A \mid p\rangle}{y\dot{\notin}A \mid \dot{\neg}p_y^x}$$

Figure 4.6: Separation Rules

It is not only the syntactic representation of a separation that can express set differences and intersections but also its semantics. Adding these rules we can omit those for differences. This indicates that the two last-mentioned operators are special cases of the first one.

Note the new structure of these rules. It is the first time that we make use of substitution in a tableau rule. This is also the point where the termination proof of the two previous systems breaks. The problem is that the substitution is capable of generating new terms that weren't initially contained in the closure. In the next section we will see how this can lead to diverging saturation attempts.

4.4.2 Divergence

Consider the following branch:

$F := \langle a\dot{\in}A \mid B\,\dot{\not\subseteq}\,\langle a\rangle\dot{\cup}C\rangle$
$x\dot{\in}F$
$B\dot{\subseteq}F$

Applying the some of the rules, we stepwise extend our branch as follows:

$x\dot{\in}A,\ \ B\,\dot{\not\subseteq}\,\langle x\rangle\dot{\cup}C$	(R1)
$y\dot{\in}B,\ \ y\,\dot{\notin}\,\langle x\rangle\dot{\cup}C$	(S3) (we write y to abbreviate $x_{B\ \langle x\rangle\dot{\cup}C}$)
$y\dot{\in}F$	(S1)
$y\dot{\in}A,\ \ B\,\dot{\not\subseteq}\,\langle y\rangle\dot{\cup}C$	(R1)

The freshly generated y found its way, via the separation F, into a substitution and produced thereby a negated subset relation with a new composite set variable. From this relation we could generate a fresh z and apply the same rules again. Thus, we never end up with a fully saturated branch.

Nevertheless there are still a lot of cases where a branch containing separations still can be saturated in a finite number of steps. We also tried to find some requirements for the branch to allow a finite saturation. A rather restrictive approach would be to allow the predicate to hold only formulas one level below the level of the separation. But, this would basically restrict separation to differences and intersections. Although we didn't find any interesting requirements of this kind, there has been done some work in this area. The authors of [7], for example, were confronted with a similar problem for their restricted universal quantifier in the untyped setting.

4.5 Downwards Compatibility of the Systems

Note that the rules in Figures 4.1, 4.2, 4.3, 4.4 and 4.5 don't introduce any new operators that weren't already in their premises. So, during the saturation process, every successor branch has exactly the same operations as the initial one.

Hence, the full system started on a branch without separations behaves exactly like the calculus with powersets and started on a branch without separations and powersets exactly like the basic system. Moreover, the separation rules applied to differences and intersections in their representations as separations behave exactly like the particular rules for these operators. Thus, if we restrict our separations to these two forms, we won't have any issues with termination.

For this reason it suffices to provide an implementation for the full system. The above argumentation allows us to use the proven properties of the smaller systems when we run the implementation on them.

Chapter 5

Implementation

In this chapter we will explain the ideas and design decisions behind the implementation of our calculi. We provide a saturation strategy in Ltac for the full system (i.e. the basic calculus with powerset and separation extensions). As we discussed in Section 4.5, this strategy also applies directly to the smaller fragments and inherits the properties we have proven for them.

5.1 Why Ltac?

We want to be able to deal with separations and implement therefore the full system. For this calculus, however, we have seen in Section 4.4.2 that it is not always possible to saturate a branch in finitely many steps. Since the decider function in a proof by reflection is formulated in Gallina, it must terminate on every input. The only way to bring a saturation procedure into this form, is to limit the expansion depth. We decided against this approach. Instead, we implement the proof search directly in Ltac. This way, we don't have to guarantee termination. Another advantage is that in pure Ltac we can use the goal management system to keep track of the branches. We can match on goals to look for the premisses of our saturation rules. This is likely to be faster than searching a branch representation for literals of a given form in Gallina.

5.2 Structure

The development is divided into four parts: The saturation rules as lemmas, an Ltac pattern matching on the goal, a number of preprocessing and optimization steps and several variants of the final automation tactic.

First we have the saturation rules of the Figures 4.1–4.6 (except for the difference rules as they can be expressed as separations) as lemmas in Coq. In the following we will refer to them as *saturation lemmas*. The fact that they are provable assures the soundness of our ruleset.

Next are the saturation patterns. Written in Ltac, their task is to find an applicable saturation rule, to pose its conclusion as an assertion and to prove it using the corresponding saturation lemma. Altogether, there are four saturation patterns. They are grouped by the structure of their conclusions. This enables a certain control over the saturation flow. The first group contains the

three branch closing rules. They are cheap in application and used to close the subgoals representing our branches. The second group is responsible for the saturation of the non-branching rules. Lastly we have the groups for the branching and the cut rules. Although cut rules have also disjunctive conclusions, for efficiency reasons we don't count them as branching rules but as a group on their own.

The next step of our development are the auxiliary tactics for normalizing the goal and for optimization. In the preprocessing, we introduce all premises. Then we negate the conclusion twice and introduce it, too. The resulting goal has then a number of literals in the assumptions and a `False` in the claim. Such a goal is provable if and only if there is a contradiction in the assumptions. As we have defined several structures via others (e.g. $A \dot{\subset} B \Leftrightarrow A \dot{\subseteq} B \dot{\wedge} B \dot{\not\subseteq} A$), now is the time to rewrite these definitions. After these steps we are in a setting where we can apply our tableau-based reasoning.

As our tactic is meant to finish a goal, we can thin out the assumptions by clearing all those that we cannot use for saturation. This step helps keeping the search space small. We can further optimize the proof search procedure by calling the `subst`-tactic periodically. It has not only a positive effect on the search space but may also replace the application of one ore more saturation rules like (S6) or (S7).

Plugging together the steps described above we obtain the following tactics:

- `fset_nocut:`

  1. Introduce all of the premises and the negated conclusion.
  2. Remove all assumptions that cannot be used for saturation.
  3. Rewrite the composite set operations and relations.
  4. Check the branch for contradictions. If they exist, close the branch and you are done. Otherwise, saturate it with the non-branching rules.
  5. Add a Coq-equality for every set equivalence (as explained earlier, fsets are extensional) and invoke `subst`.
  6. Check again for contradictions. If there are none, apply exactly one branching rule and proceed with step 4 in every subbranch.

  The tactic terminates if it has closed all branches or if none of the branching and non-branching rules is applicable anymore.

- `fset_dec` performs the same steps as `fset_nocut` but doesn't terminate if none of the branching and non-branching rules is applicable. Instead, it applies a cut rule then and proceeds with step 4.

- `fset_decu` takes the name of a definition or a tuple of such as argument and unfolds all of them before invoking `fset_dec`. It is defined as follows.

Listing 5.1: fset_decu Pseudocode

```
1       fset_decu def :=
2       unfold def in *; fset_dec.
3       fset_decu (def_1, ..., def_n) :=
4       unfold def_1 in *; ...; unfold def_n in *;
5       fset dec.
```

- `fset_nocutu` does the same as `fset_decu` but applies `fset_nocut` at the end instead of `fset_dec`.

Remark. The above is an informal description of the ideas implemented in our automation tactics. For simplicity some of the steps of the submitted implementation where omitted in this description.

5.3 Examples

In this section we will show some interesting examples for propositions that can be proven by one of the presented tactics. The code of these and other examples can be found in the file tabtest.v.

Example. a) In the environment

$$C := \langle X \dot{\in} \dot{\mathcal{P}}(B) \mid X \dot{\subseteq} \langle y \dot{\in} X \mid py \rangle \rangle$$

the proposition

$$A \dot{\subseteq} B \to \langle x \dot{\in} A \mid px \rangle \dot{\in} C$$

is proven instantaneously.

b) The propositions

$$A \dot{\subseteq} C \to B \dot{\subseteq} C \to ((C \dot{-} A) \dot{\cup} (C \dot{-} B)) \doteq (C \dot{-} (A \dot{\cap} B))$$

$$A \dot{\subseteq} C \to B \dot{\subseteq} C \to ((C \dot{-} A) \dot{\cap} (C \dot{-} B)) \doteq (C \dot{-} (A \dot{\cup} B))$$

are proved either in less than half a second.

c) The proposition

$$\dot{\mathcal{P}}(A \dot{\cup} B) \dot{\subseteq} \dot{\mathcal{P}}(A) \dot{\cup} \dot{\mathcal{P}}(B) \to A \dot{\subseteq} B \dot{\vee} B \dot{\subseteq} A$$

which requires application of cut rules is solved in about 7 seconds.

d) The example for the necessity of cuts in the basic ruleset in Section 4.2.1

$$A \dot{-} B \dot{\subseteq} \dot{\emptyset} \to x \dot{\in} A \to x \dot{\in} B$$

is solved instantaneously.

e) The example for the importance of cut rules in the powerset extension

$$\dot{\mathcal{P}}(A) \dot{\subseteq} \dot{\mathcal{P}}(B) \to A \dot{\subseteq} B$$

is solved in 2.73 seconds.

f) We had in Section 4.3.3 the branch

$$\Gamma := \{\dot{\mathcal{P}}(A) \dot{-} \langle A \rangle \dot{\subseteq} \dot{\emptyset}, \quad x \dot{\in} A\}$$

as an example for the possible incompleteness of the calculus since we were not able to close it. However, if we replace the literal $x \dot{\in} A$ by $A \dot{\neq} \dot{\emptyset}$, i.e. we prove

$$\dot{\mathcal{P}}(A) \dot{-} \langle A \rangle \dot{\subseteq} \dot{\emptyset} \to A \doteq \dot{\emptyset}$$

the tactic succeeds after less than one second.

g) The following goal is representative for a class of problems that is trivial but tedious to prove by hand. In this or a similar form it could emerge in the area of metatheory of model logic [13]. `fset_decu` solves it instantaneously.

```
Variables (T : choiceType)
  (p q r : pred {fset T})
  (F : {fset T}).
Definition U := powerset F.
Definition S0 := [fset D in U | p D && q D].
Definition A := [fset D in S0 | r D].

Lemma test1 s C : s \in C -> C \in A -> s \in F.
Proof. fset_decu (A,S0,U). Qed.
```

Bibliography

[1] Adam Chlipala: *Certified Programming with Dependent Types (2014).* `http://adam.chlipala.net/cpdt/`

[2] Samuel Boutin: Using Reflection to Build Efficient and Certified Decision Procedures. TACS 1997: 515-529, Springer 1997

[3] Christian Doczkal: *Finite Sets over Countalbe Types in Ssreflect* `http://www.ps.uni-saarland.de/formalizations/fset.php`

[4] Alfredo Ferro, Eugenio G. Omodeo, Jacob T. Schwartz: *Decision procedures for some fragments of set theory.* CADE 1980: 88-96, Springer 1980

[5] Domenico Cantone, Calogero G. Zarba: *A New Fast Tableau-Based Decision Procedure for an Unquantified Fragment of Set Theory.* FTP (LNCS Selection) 1998: 126-136, Springer 2000

[6] Bernhard Beckert, Ulrike Hartmer: *A Tableau Calculus for Quantifier-Free Set Theoretic Formulae.* TABLEAUX 1998: 93-107, Springer 1998

[7] Domenico Cantone, Calogero G. Zarba: *A Tableau-Based Decision Procedure for a Fragment of Set Theory Involving a Restricted Form of Quantification.* TABLEAUX 1999: 97-112, Springer 1999

[8] Domenico Cantone: *Decision procedures for elementary sublanguages of set theory: X. Multilevel syllogistic extended by the singleton and powerset operators.* J. Autom. Reasoning 7:193-230, 1991, Springer 1991

[9] Domenico Cantone, Rosa Ruggeri Cannata: Deciding set-theoretic formulae with the predicate 'finite' by a tableau calculus. Le Matematiche Vol 50, No 1 (1995)

[10] Domenico Cantone, Calogero G. Zarba, Rosa Ruggeri Cannata: *A Tableau-Based Decision Procedure for a Fragment of Set Theory with Iterated Membership.* J. Autom. Reasoning 34(1): 49-72 (2005), Springer 2005

[11] Benjamin Shults: *Comprehension and Description in Tableaux.* 1997

[12] Coq Development Team: *Coq Documentation* `https://coq.inria.fr/documentation`

[13] Christian Doczkal, Gert Smolka Completeness and Decidability Results for CTL in Coq Interactive Theorem Proving (ITP 2014), Vol. 8558 of LNAI, pp. 226-241, Springer, 2014